Chapter THIRTEEN
Iterating and Filtering Collections

Exam Objectives

Collections Streams and Filters.
Iterate using forEach methods of Streams and List.
Filter a collection by using lambda expressions.

Iteration

Usually, when you have a list, you'd want to iterate over its elements. A common way is to use a `for` block.

Either with an index:

```
List<String> words = ...
for(int i = 0; i < words.size(); i++) {
    System.out.println(words.get(i));
}
```

Or with an iterator:

```
List<String> words = ...
for(Iterator<String> it = words.iterator(); it.hasNext();) {
    System.out.println(it.next());
}
```

Or with the so-called `for-each` loop:

```
List<String> words = ...
for(String w : words) {
    System.out.println(w);
}
```

Besides looking ugly, the first two add points where an error can happen (the index and iterator variables). The recommended way is to use the `for-each` loop whenever you can.

Therefore, taking advantage of functional interfaces, Java 8 adds another option to iterate lists based on the `for-each` loop, the `foreach` method:

```
default void forEach(Consumer<? super T> action)
```

Since this method is defined in the `Iterable` interface, is not only available to lists, but to all the implementations of this interface, such as `Queues`, `Sets`, `Deques`, and even some SQL-related exceptions, like `SQLException`.

Also, notice this is a default method, which means that there's a default implementation that implementing classes can override (and many do, mostly to deal with concurrent modifications).

This is the default implementation:

```
for (T t : this) {
    action.accept(t);
}
```

So basically, it's a `for-each` loop using the new functional features of Java 8.

To use it, we can start with an anonymous class:

```
List<String> words = ...
words.forEach(new Consumer<String>() {
    public void accept(String t) {
        System.out.println(t);
    }
});
```

Remember that the `Consumer` interface represents an operation that takes one parameter but doesn't return any result.

That anonymous class can be transformed into a lambda expression:

```
words.forEach(t -> System.out.println(t));
```

Or in this particular example, a method reference:

```
words.forEach(System.out::println);
```

Just remember the rules about using final or effectively final variables inside anonymous classes or lambda expressions. Code like the following is not valid:

```
int max = 0;
words.forEach(t -> {
    // The following line won't compile, you can't modify max
    max = Math.max(max, t.length());
    System.out.println(t);
});
```

If you want to do things like this (get the max length of all the strings in a list), it's better to use streams to iterate the collection and apply other operations (for this particular example, we'll see how to calculate the max length with the `reduce` method in a later chapter).

The `Stream` interface provides a corresponding `forEach` method:

```
void forEach(Consumer<? super T> action)
```

Since this method doesn't return a stream, it represents a terminal operation.

Using it is not different than the `Iterable` version:

```
Stream<String> words = ...
// As an anonymous class
words.forEach((new Consumer<String>() {
    public void accept(String t) {
        System.out.println(t);
    }
});
// As a lamba expression
words.forEach(t -> System.out.println(t));
// As a method reference
words.forEach(System.out::println);
```

Of course, the advantage of using streams is that you can chain operations, for example:

```
words.sorted()
    .limit(2)
    .forEach(System.out::println);
```

As a terminal operation, you can't do things like this:

```
words.forEach(t -> System.out.println(t.length()));
words.forEach(System.out::println);
```

If you want to do something like this, either create a new stream each time:

```
Stream.of(wordList).forEach(t -> System.out.println(t.length()));
Stream.of(wordList).forEach(System.out::println);
```

Or wrap the code inside one lambda:

```
Consumer<String> print = t -> {
    System.out.println(t.length());
    System.out.println(t);
};
words.forEach(print);
```

You can't use `return`, `break` or `continue` to terminate an iteration either. `break` and `continue` will generate a compilation error since they cannot be used outside of a loop and return doesn't make sense when we see that the `foreach` method is implemented basically as:

```
for (T t : this) {
   // Inside accept, return has no effect
   action.accept(t);
}
```

As a side note (since it's not covered in the exam), Java 8 also added a `forEach` method to the `Map` interface. However, since a map has a key and value, this new method takes a `BiConsumer`:

```
default void forEach(BiConsumer<? super K,? super V> action)
```

With a default implementation is equivalent to:

```
for (Map.Entry<K, V> entry : map.entrySet()) {
    action.accept(entry.getKey(), entry.getValue());
}
```

Filtering

Another common requirement is to filter (or remove) elements from a collection that don't match a particular condition.

You normally do this either by copying the matching elements to another collection:

```
List<String> words = ...
List<String> nonEmptyWords = new ArrayList<String>();
for(String w : words) {
    if(w != null && !w.isEmpty()) {
        nonEmptyWords.add(w);
    }
}
```

Or by removing the non-matching elements in the collection itself with an iterator (only if the collection supports removal):

```
List<String> words = new ArrayList<String>();
// ... (add some strings)
for (Iterator<String> it = words.iterator(); it.hasNext();) {
    String w = it.next();
    if (w == null || w.isEmpty()) {
        it.remove();
    }
}
```

Now in Java 8, there's a new method on the `Collection` interface:

```
default boolean removeIf(Predicate<? super E> filter)
```

That removes all of the elements of the collection that satisfy the given predicate (the default implementation uses the iterator version).

This makes the code simpler by using either lambda expressions or method references:

```
// Using an anonymous class
words.removeIf(new Predicate<String>() {
    public boolean test(String t) {
        return t == null || t.isEmpty();
    }
});
// Using a lambda expression
words.removeIf(t -> t == null || t.isEmpty());
```

For the case where you copy the matching elements to another collection, you have the `filter` method of the `Stream` interface:

```
Stream<T> filter(Predicate<? super T> predicate)
```

That returns a new stream consisting of the elements that satisfy the given predicate.

Since this method returns a stream, it represents an intermediate operation, which basically means that you can chain any number of filters or other intermediate operations:

```
List<String> words = Arrays.asList("hello", null, "");
words.stream()
    .filter(t -> t != null) // ["hello", ""]
    .filter(t -> !t.isEmpty()) // ["hello"]
    .forEach(System.out::println);
```

Of course, the result of executing this code is:

```
hello
```

You can also create a method (or use an existing one) on some class to do it with a method reference, for the sole purpose of clarity:

```
class StringUtils {
    public static boolean isNotNullOrEmpty(String s) {
        return s != null && !s.isEmpty();
    }
}
// ...
List<String> words = Arrays.asList("hello", null, "");
// Using an anonymous class
words.stream()
    .filter(new Predicate<String> () {
        public boolean test(String t) {
            return StringUtils.isNotNullOrEmpty(t);
        }
    })
    .forEach(System.out::println);
// Using a lambda expression
words.stream()
    .filter(t -> StringUtils.isNotNullOrEmpty(t))
    .forEach(System.out::println);
// Using a lambda expression
words.stream()
    .filter(StringUtils::isNotNullOrEmpty)
    .forEach(System.out::println);
```

The `Stream` interface also has the `distinct` method to filter duplicate elements, according to the `Object.equals(Object)` method.

```
Stream<T> distinct()
```

Again, since it returns a new stream, this is an intermediate operation. Because it has to know the values of the elements to find out which ones are duplicates, this operation is also stateful.

Here's an example:

```
List<String> words = Arrays.asList("hello", null, "hello");
words.stream()
    .filter(t -> t != null) // ["hello", "hello"]
    .distinct() // ["hello"]
    .forEach(System.out::println);
```

Of course, the result is:

```
hello
```

Key Points

- Java 8 adds the following method to the `Iterable` interface as another option to iterate the implementations of this interface (like lists):

```
default void forEach(Consumer<? super T> action)
```

- For example:

```
List<String> words = Arrays.asList("hello", "world");
words.forEach(t -> System.out.println(t));
```

- The `Stream` interface also has this method:

```
void forEach(Consumer<? super T> action)
```

- This is a terminal operation. Here's an example:

```
Stream<String> words = Stream.of("hello", "world");
words.forEach(t -> System.out:println(t));
```

- Of course, the advantage of using streams is that you can chain operations, for example:

```
words.sorted()
    .limit(2)
    .forEach(System.out::println);
```

- But as a terminal operation, you can't do things like this:

```
words.forEach(t -> System.out.println(t.length()));
words.forEach(System.out::println);
```

- For filtering, on the side of collections, we have a new method:

```
default boolean removeIf(Predicate<? super E> filter)
```

- That removes all of the elements of the collection that satisfy the given predicate.
- On the `Stream` interface, we have:

```
Stream<T> filter(Predicate<? super T> predicate)
```

- That returns a new stream consisting of the elements that satisfy the given predicate.
- Since this method returns a stream, it represents an intermediate operation, which means that you can chain any number of filters or other intermediate operations:

```
List<String> words = Arrays.asList("hello", null, "");
words.stream()
    .filter(t -> t != null) // ["hello", ""]
    .filter(t -> !t.isEmpty()) // ["hello"]
    .forEach(System.out::println);
```

- The `Stream` interface also has the distinct method to filter duplicate elements, according to the `Object.equals(Object)` method.

```
Stream<T> distinct()
```

- This is an intermediate operation, and because it has to know the values of the elements to find out which ones are duplicates, this operation is also stateful. Here's an example:

```
List<String> words = Arrays.asList("hello", null, "hello");
words.stream()
    .filter(t -> t != null) // ["hello", "hello"]
    .distinct() // ["hello"]
    .forEach(System.out::println);
```

Self Test

1. Given:

```
public class Question_13_1 {
    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,2,3,4,5,6);
        Stream.of(l)
            .forEach(i -> System.out.print(i-1));
    }
}
```

What is the result?
A. `123456`
B. `012345`
C. `543210`
D. Compilation error
E. An exception is thrown

2. Which of the following statements is true?
A. `filter` is a terminal operation.
B. `filter` is a stateful operation.
C. `Stream.forEach` takes an implementation of the `Consumer` functional interface as an argument.
D. You can chain more than one `forEach` operation in a stream pipeline.

3. Given:

```
public class Question_13_2 {
    public static void main(String[] args) {
        Arrays.asList(1,2,3,4,5,6).stream()
            .filter(i -> i%2 == 0).filter(i -> i > 3)
            .forEach(System.out::print);
    }
}
```

What is the result?
A. `246`
B. `46`
C. `1`
D. `5`
E. Compilation error

4. Given:

```
public class Question_13_3 {
    public static void main(String[] args) {
        Arrays.asList(1,1,1,1,1,1).stream()
            .filter(i -> i > 1).distinct()
            .forEach(System.out::print);
    }
}
```

What is the result?
A. `1`
B. `0`
C. Nothing is printed
D. Compilation error
E. An exception is thrown

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

12. Streams 14. Optional Class